



Lecture 2

Constrained minimization problem

Constrained minimization problem

Let us consider the problem of nonlinear programming in general form:

$$\begin{aligned} f(x) &\rightarrow \min(\max) \\ g_i(x) &\leq 0, i = 1, \dots, m \\ h_i(x) &= 0, i = 1, \dots, k \end{aligned}$$

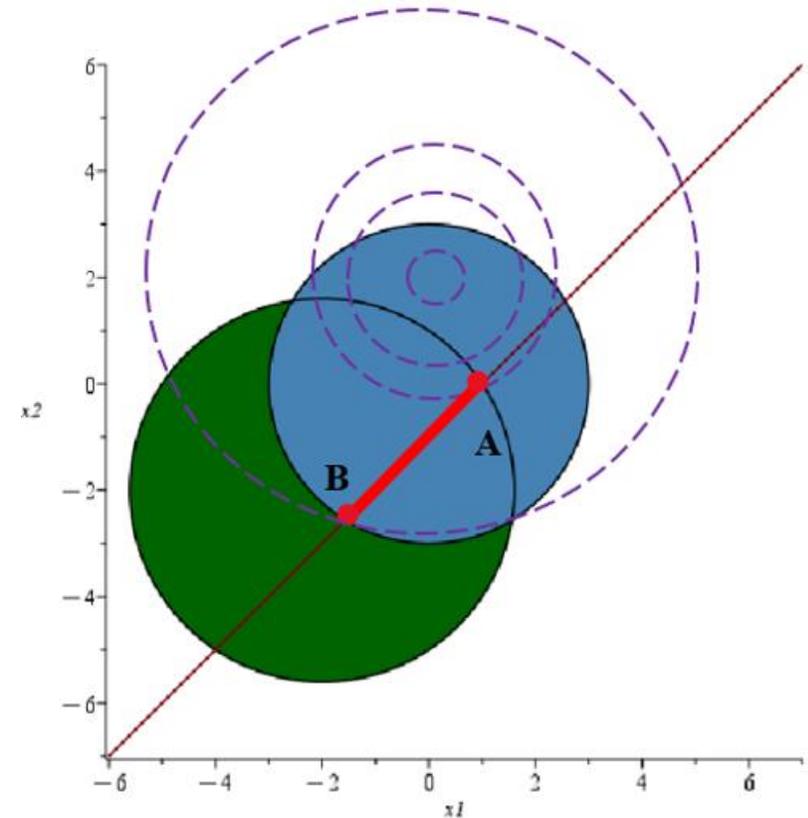
For example, task:

$$\begin{cases} f(x) = x_1^2 + x_2^2 - 4x_2 \rightarrow \min \\ x_1^2 + x_2^2 \leq 9 \\ x_1^2 + 4x_1 + x_2^2 + 4x_2 \leq 5 \\ x_1 - x_2 = 1 \end{cases}$$

Constrained minimization problem. Graphic solution

In the case of two variables, the problem can be solved graphically. It is necessary to show the constraints and level lines of the objective function

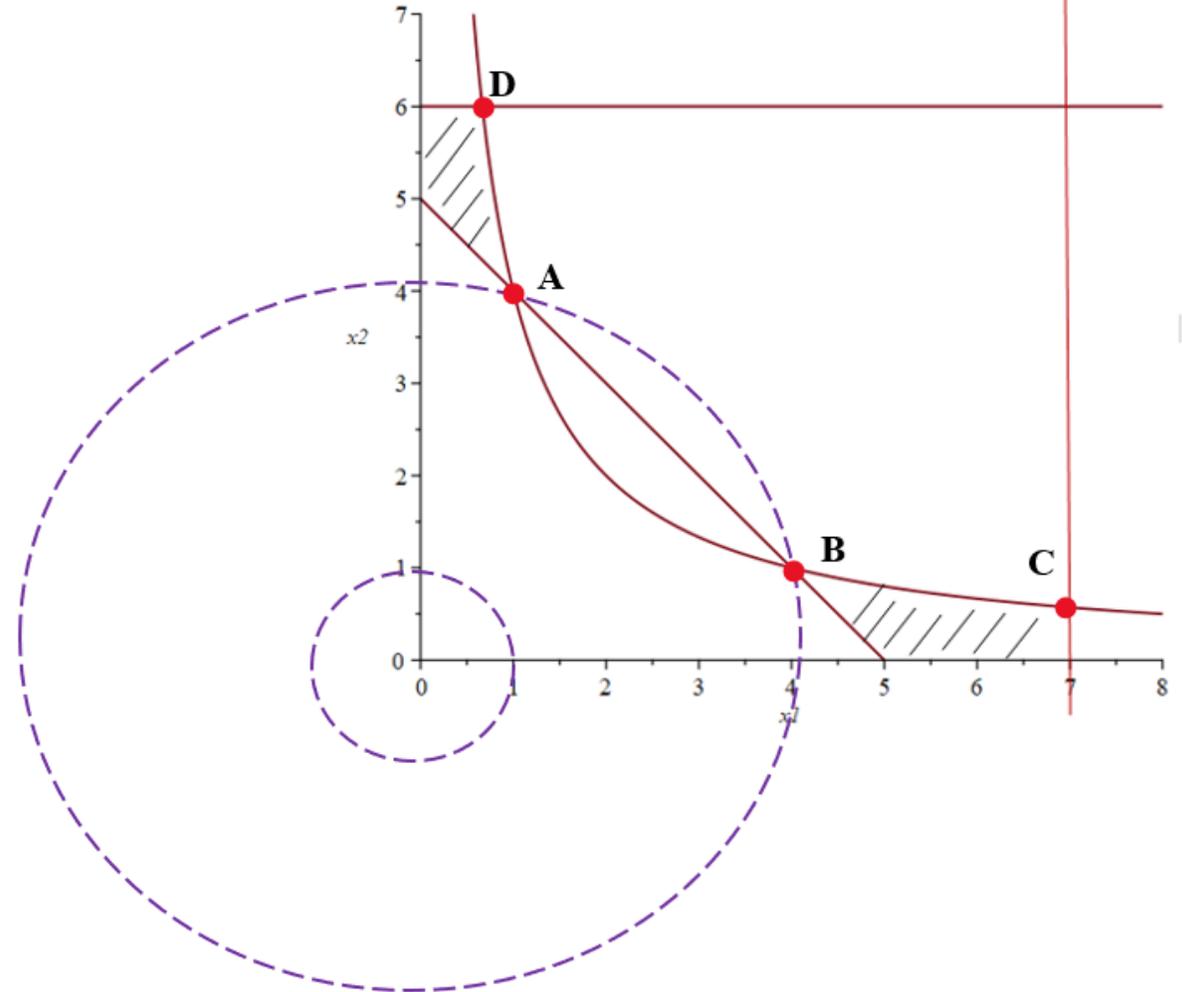
$$\begin{cases} f(x) = x_1^2 + x_2^2 - 4x_2 \rightarrow \min \\ x_1^2 + x_2^2 \leq 9 \\ x_1^2 + 4x_1 + x_2^2 + 4x_2 \leq 5 \\ x_1 - x_2 = 1 \end{cases}$$



Constrained minimization problem. Graphic solution

$$f(x) = x_1^2 + x_2^2 \rightarrow \min$$

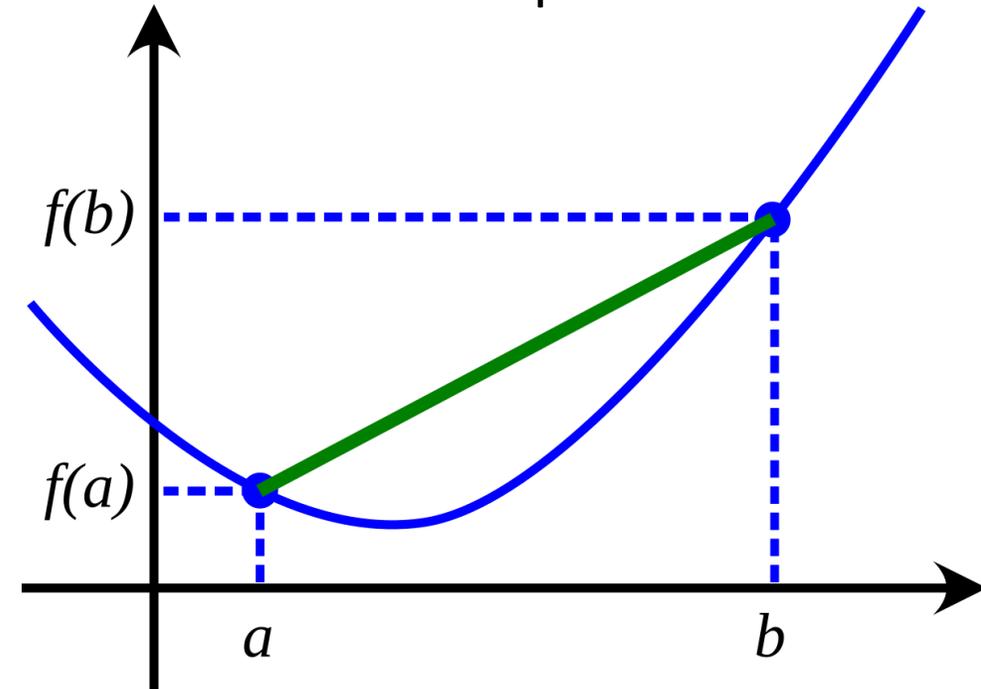
$$\begin{cases} x_1 x_2 \leq 4 \\ x_1 + x_2 \geq 5 \\ x_1 \leq 7 \\ x_2 \leq 6 \\ x \geq 0 \end{cases}$$



Convex functions. Testing a function for convexity

Nonlinear programming problems with convex functions are well studied and belong to the section of convex programming, so let's start with the concept of a convex function.

In mathematics, a real-valued function is called **convex** if the line segment between any two distinct points on the graph of the function lies above or on the graph between the two points



Convex functions. Testing a function for convexity

Algorithm for checking a function for convexity.

1. Construct the Hessian matrix for the function in question.
2. Calculate the principal minors of the Hessian matrix. If all principal minors are non-negative, then conclude that the function is convex.

The principal minor of the k -th order is the determinant composed of elements of a matrix of quadratic form located at the intersection of k rows and k columns of the given matrix with the same numbers.

A **principal minor** is the determinant of a square submatrix obtained by deleting a set of rows **and the corresponding set of columns** (i.e., the same indices).

Convex programming problem

$$\begin{cases} f(x) \rightarrow \min \\ g_i(x) \leq 0, i = 1, \dots, m \\ x \geq 0 \end{cases}$$

where $f(x)$ and $g_i(x), i = 1, \dots, m$ are convex

The **Lagrange function** of a convex programming problem is the function

$$L(x, y) = f(x) + \sum_{i=1}^m g_i(x)\lambda_i$$

where $\lambda_i, i = 1, \dots, m$ – Lagrange multipliers

Convex programming problem

A point (x^*, λ^*) , $x^* \geq 0, \lambda^* \geq 0$ is called a **saddle point of the Lagrange function** if:

$$L(x^*, \lambda) \leq L(x^*, \lambda^*) \leq L(x, \lambda^*), \forall x, \lambda \geq 0$$

1st Kuhn-Tucker theorem. If (x^*, λ^*) is a saddle point of the Lagrange function, then x^* is an optimal solution to the convex programming problem.

2nd Kuhn-Tucker theorem. For any optimal solution x^* of a regular convex programming problem, there exists a vector of Lagrange multipliers λ^* : (x^*, λ^*) is a saddle point of the Lagrange function.

Nonlinear programming problem with equality constraints

Let us consider a problem of conditional optimization with equality constraints of the form:

$$\begin{cases} f(x) \rightarrow \max(\min) \\ g_i(x) = 0, i = 1, \dots, m \end{cases}$$

where $f(x)$, $g_i(x)$, $i = 1, \dots, m$ continuously differentiable functions.

In this method, the main idea of the solution is to find the unconditional extremum of a specially constructed Lagrange function. And then the classical method for solving the problem of unconditional minimization is used.

Nonlinear programming problem with equality and inequality constraints. Lagrange multiplier method

Let us consider a problem of conditional optimization with equality constraints and inequalities of the form:

$$\begin{cases} f(x) \rightarrow \max(\min) \\ g_i(x) = 0, i = 1, \dots, m_1 \\ g_i(x) \leq 0, i = m_1 + 1, \dots, m \end{cases}$$

where $f(x)$, $g_i(x)$, $i = 1, \dots, m$ continuously differentiable functions.



SciPy

Contains a large set of functions for scientific computing, including tools for solving optimization problems, which are located in the **scipy.optimize** module.

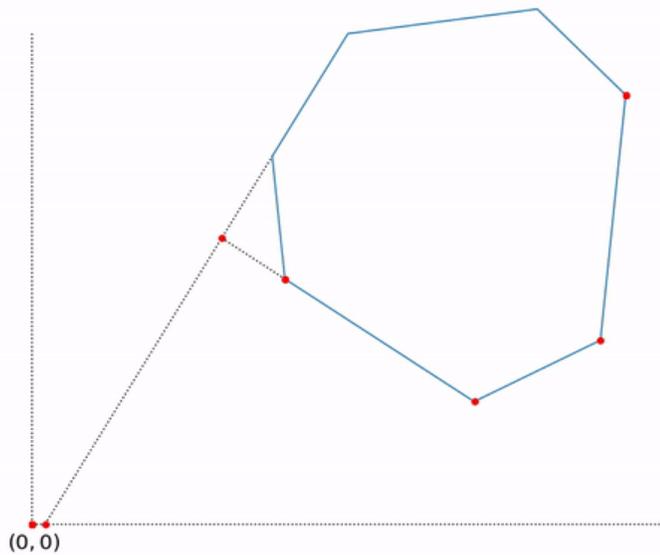
This module provides methods for solving both nonlinear programming (NLP) and linear programming (LP) problems, including mixed integer linear programming (MILP) problems.

SciPy. Solving the MILP problem

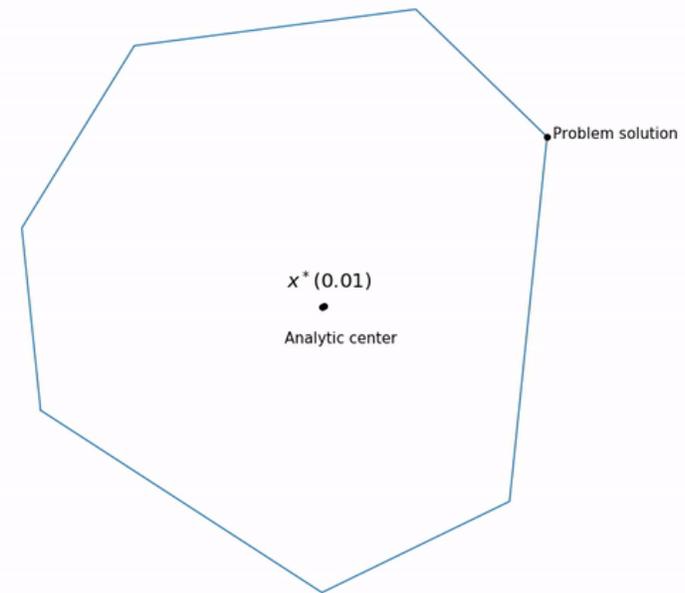
To solve linear programming problems, the optimize submodule has a **linprog** function. **HiGHS** is used by default as a solver for **LP(MILP)** — it implements the simplex method (**highs-ds**) and the interior point method (**highs-ipm**). When starting the solution, one of the methods is selected by default; the ability to force a method is also present.

Interior point methods (highs-ipm) work exceptionally well on very large but sparse problems (where the constraint matrices contain many zeros). The simplex method (highs-ds) can be more efficient on medium-sized problems or those with a denser structure.

SciPy. Solving the MILP problem



simplex method



the interior point method

SciPy. Solving the MILP problem

$$\begin{aligned}f(x) &= 3x_1 + 3x_2 \rightarrow \max \\x_1 + x_2 &\leq 8 \\2x_1 - x_2 &\geq 1 \\x_1 - 2x_2 &\leq 2 \\x_1 &\geq 0, x_2 \geq 0\end{aligned}$$

```
from scipy.optimize import linprog

c = - np.array([3., 3.])
A_ub = np.array([[1., 1.], [-2., 1.], [1., -2.]])
b_ub = np.array([8.0, -1.0, 2.0])

bounds = [(0, np.inf), (0, np.inf)]
res_milp = linprog(c=c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

print(f"Solution: x = {list(np.round(res_milp['x'], 2))}, f = {-res_milp['fun']}, {res_milp['message']}")
```

Solution: x = [3.0, 5.0], f = 24.0, Optimization terminated successfully. (HiGS Status 7: Optimal)

Conditional optimization method="trust-constr"



The **trust-constr method** in the minimize function of the **scipy.optimize** library is a constrained optimization method based on trust-regions. It is designed to solve optimization problems with constraints (both linear and nonlinear) and can work with both continuous and discrete variables.

Algorithm (Trust Region).

Given $\bar{\Delta} > 0$, $\Delta_0 \in (0, \bar{\Delta})$, and $\eta \in [0, \frac{1}{4})$:

for $k = 0, 1, 2, \dots$

 Obtain p_k

 Evaluate ρ_k

if $\rho_k < \frac{1}{4}$

$$\Delta_{k+1} = \frac{1}{4} \|p_k\|$$

else

if $\rho_k > \frac{3}{4}$ and $\|p_k\| = \Delta_k$

$$\Delta_{k+1} = \min(2\Delta_k, \bar{\Delta})$$

else

$$\Delta_{k+1} = \Delta_k;$$

if $\rho_k > \eta$

$$x_{k+1} = x_k + p_k$$

else

$$x_{k+1} = x_k;$$

end (for).

This shows how well the model predicted the actual decrease in function.

$\bar{\Delta}$ – maximum radius of the trust region

Δ_0 – initial radius

$$\min m_k(p) \quad \|p\| \leq \Delta_k$$

Conditional optimization method="trust-constr"

$$f(x) = 3x_1 + 3x_2 \rightarrow \min$$

$$x_1 + x_2 \leq 8$$

$$2x_1 - x_2 \geq 1$$

$$x_1 - 2x_2 \leq 2$$

$$x_1 \geq 0, x_2 \geq 0$$

Number of iterations: 229, function evaluations: 615, CG iterations: 212, optimality: 1.29e-09, constraint violation: 3.00e+00, execution time: 0.37 s.
Solution: x = [3.199999999]
Value: 14.999999984629042

```
from scipy.optimize import minimize, LinearConstraint, Bounds
import numpy as np

bounds = Bounds([0, 0], [np.inf, np.inf])

linear_constraint = LinearConstraint(
    [[1, 1], [-2, 1], [1, -2]],
    [8, -1, 2],
    [np.inf, np.inf, np.inf]
)

def objective(x):
    return 3 * x[0] + 3 * x[1]

x0 = np.array([5, 5])

res = minimize(
    objective,
    x0,
    method='trust-constr',
    constraints=[linear_constraint],
    bounds=bounds,
    options={'verbose': 1}
)

print(f"Solution: x = {res.x}")
print(f"Value: {res.fun}")
```

CVXPY. Overview



CVXPY - this package was implemented to solve convex optimization problems. To solve the problem, it is necessary to perform several steps: define variables, set the objective function and constraints to form the object of the optimization problem. After the problem is formulated, before running the solver, the convexity of the objective function and constraints is checked using the **DCP** (Disciplined Convex Programming) rules.

```

import cvxpy as cp

x = cp.Variable()
y = cp.Variable()

objective = cp.Minimize(x**2 + y**2 - 2*x)

constraints = [
    x**2 + 4*y**2 - 4*x - 4*y <= 0,
    x + y >= 4,
    x >= 0
]

assert objective.is_dcp(), "Objective function is not convex"
for constraint in constraints:
    assert constraint.is_dcp(), "Constraint is not convex"

problem = cp.Problem(objective, constraints)
problem.solve()

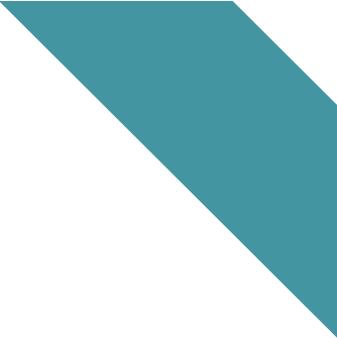
print("Minimal value:", problem.value)
print("Optimal x and y:", x.value, y.value)

```

```

Minimal value: 3.500000000367045
Optimal x and y: 2.5000000056865748 1.4999999944357738

```

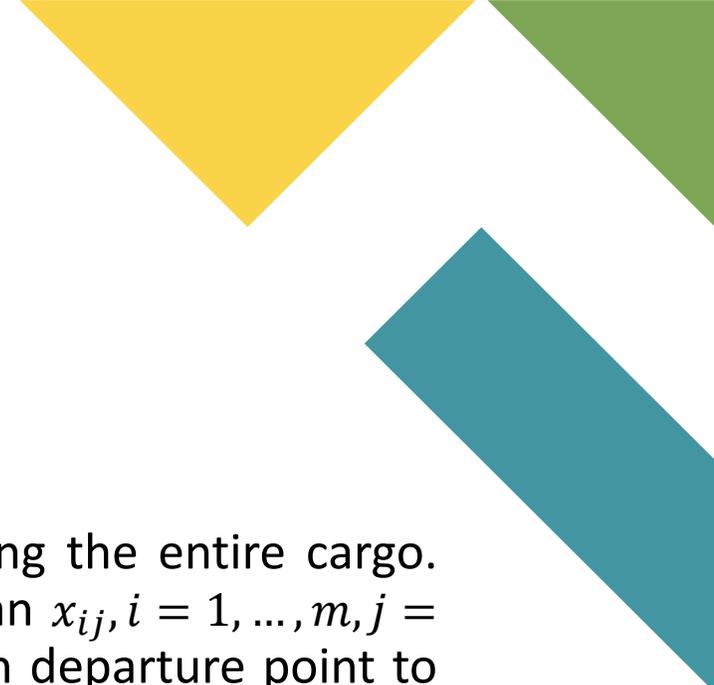
$$\begin{aligned}
 f(x) &= x_1^2 + x_2^2 - 2x_1 \rightarrow \min \\
 \begin{cases}
 x_1^2 + 4x_2^2 - 4x_1 - 4x_2 \leq 0 \\
 x_1 + x_2 \geq 4 \\
 x \geq 0
 \end{cases}
 \end{aligned}$$


Transport problem

The transportation problem is a linear programming problem and its purpose is to determine the most economical plan for transporting goods from points of departure (warehouses) to points of destination (stores).

Let there be m warehouses A_1, A_2, \dots, A_m from which goods should be transported to n stores B_1, B_2, \dots, B_n . Let $c_{ij}, i = 1, \dots, m, j = 1, \dots, n$ be the cost of transporting one unit of cargo from the i -th warehouse to the j -th store (transportation tariff). Let $a_i, i = 1, \dots, m$ be the cargo stock at the i -th warehouse; $b_j, j = 1, \dots, n$ — be the cargo demand at the j -th store; $x_{ij}, i = 1, \dots, m, j = 1, \dots, n$ — be the number of cargo units transported from the i -th departure point to the j -th destination point. The matrix $X = (x_{ij})_{m \times n}$ is called the transportation plan. The matrix $C = (c_{ij})_{m \times n}$ is called the tariff matrix.

Transport problem



As the optimality criterion, we will choose the minimum cost of transporting the entire cargo. Thus, **to solve the transport problem** is to determine the transportation plan $x_{ij}, i = 1, \dots, m, j = 1, \dots, n$, that is, the amount of cargo that should be transported from the i -th departure point to the j -th destination point so that all departure points get rid of cargo reserves, and all destination points receive the required amount of cargo, and at the same time the cost of transporting the entire cargo is the lowest.

Transport problem

The mathematical model of the transport problem will take the form:

$$F = \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \rightarrow \min$$

$$\sum_{i=1}^m x_{ij} = b_j, j = 1, \dots, n$$

$$\sum_{j=1}^n x_{ij} = a_i, i = 1, \dots, m$$

$$x_{ij} \geq 0, i = 1, \dots, m, j = 1, \dots, n$$

Transport problem. Example

Warehouses	Stores				Stocks
	B_1	B_2	B_3	B_4	
A_1	2 x_{11}	3 x_{12}	4 x_{13}	2 x_{14}	140
A_2	8 x_{21}	4 x_{22}	1 x_{23}	4 x_{24}	160
A_3	9 x_{31}	7 x_{32}	3 x_{33}	6 x_{34}	120
Needs	150	90	100	80	

Warehouses	Stores				Stocks
	B_1	B_2	B_3	B_4	
A_1	2 x_{11}	3 x_{12}	4 x_{13}	2 x_{14}	140
A_2	8 x_{21}	4 x_{22}	1 x_{23}	4 x_{24}	160
A_3	9 x_{31}	7 x_{32}	3 x_{33}	6 x_{34}	120
Needs	150	90	100	80	

```

from scipy.optimize import linprog

# objective function coefficients
c = [2, 3, 4, 2, 8, 4, 1, 4, 9, 7, 3, 6]

stocks = [140, 160, 120]
A_ub = [
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
]

```

```

needs = [150, 90, 100, 80]
A_eq = [
    [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
    [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
]

```

```

Optimal solution: [140.  0.  0.  0.  0.  90.  70.  0.  10.  0.  30.  80.]
Optimal value of the objective function: 1370.0

```

```

result = linprog(c, A_ub=A_ub, b_ub=stocks, A_eq=A_eq, b_eq=needs)

print("Optimal solution:", result.x)
print("Optimal value of the objective function:", result.fun)

```